Fundamentos de Programación II

Tema 1. Ordenación, búsqueda e intercalación interna

Luis Rodríguez Baena (luis.rodriguez@upsam.es)

Universidad Pontificia de Salamanca
Escuela Superior de Ingeniería y Arquitectura

Ordenación interna

- Reorganización de un conjunto dado de objetos en una secuencia especificada.
  - Permutará las posiciones de los elementos de forma que sus claves formen una secuencia creciente.
  - La ordenación se suele realizar sobre un conjunto de registros con un campo clave que identifique el registro.
    - ✓ Dados los registros r_1, r_2,..., r_n con valores de clave k_1, k_2,..., k_n debe resultar la misma secuencia de registros en orden r_{i_1}, r_{i_2},..., r_{i_n} tal que $k_{i_1} \leq k_{i_2} \leq \ldots k_{i_n}$.
  - Una clave o un campo clave es el dato a partir del cual se va a realizar la ordenación.
    - ✓ Por ejemplo, si queremos ordenar una lista de personas a partir de su DNI, el campo clave será el DNI.
- Los métodos se pueden clasificar según su eficiencia.
  - La eficiencia se puede medir en base al número de comparaciones y movimientos que realizan.
    - ✓ Métodos directos (n^2 comparaciones de claves).
    - ✓ Métodos no directos ($n \log n$ comparaciones de claves).

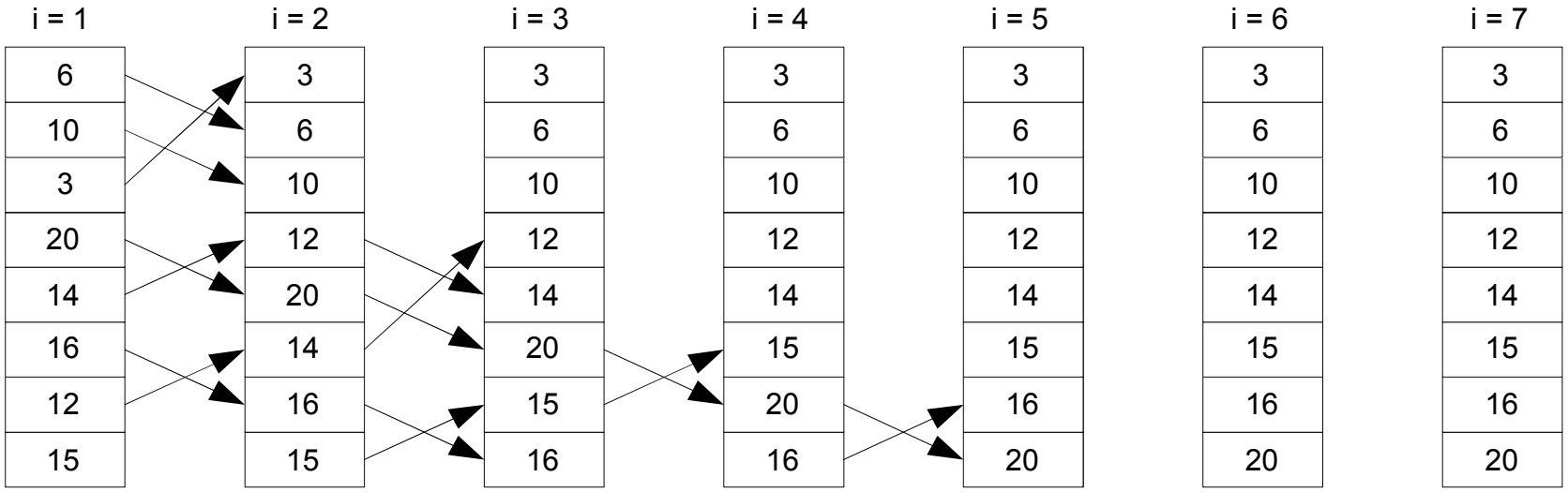
Ordenación interna (II)

- Para todos los métodos se va a suponer que exixten estas declaraciones iniciales…

```
const
   numElementos = ... //Número de elementos del array a ordenar
tipos
   //tipoElemento podrá ser cualquier tipo de dato simple, por ejemplo
   entero = tipoElemento
   array[0..numElementos] de tipoElemento = vector
```

```
procedimiento intercambiar(ref tipoElemento : a,b)
var
   tipoElemento : aux
inicio
   aux ← a
   a ← b
   b ← aux
fin_procedimiento
```

Ordenación por intercambio directo (burbuja)



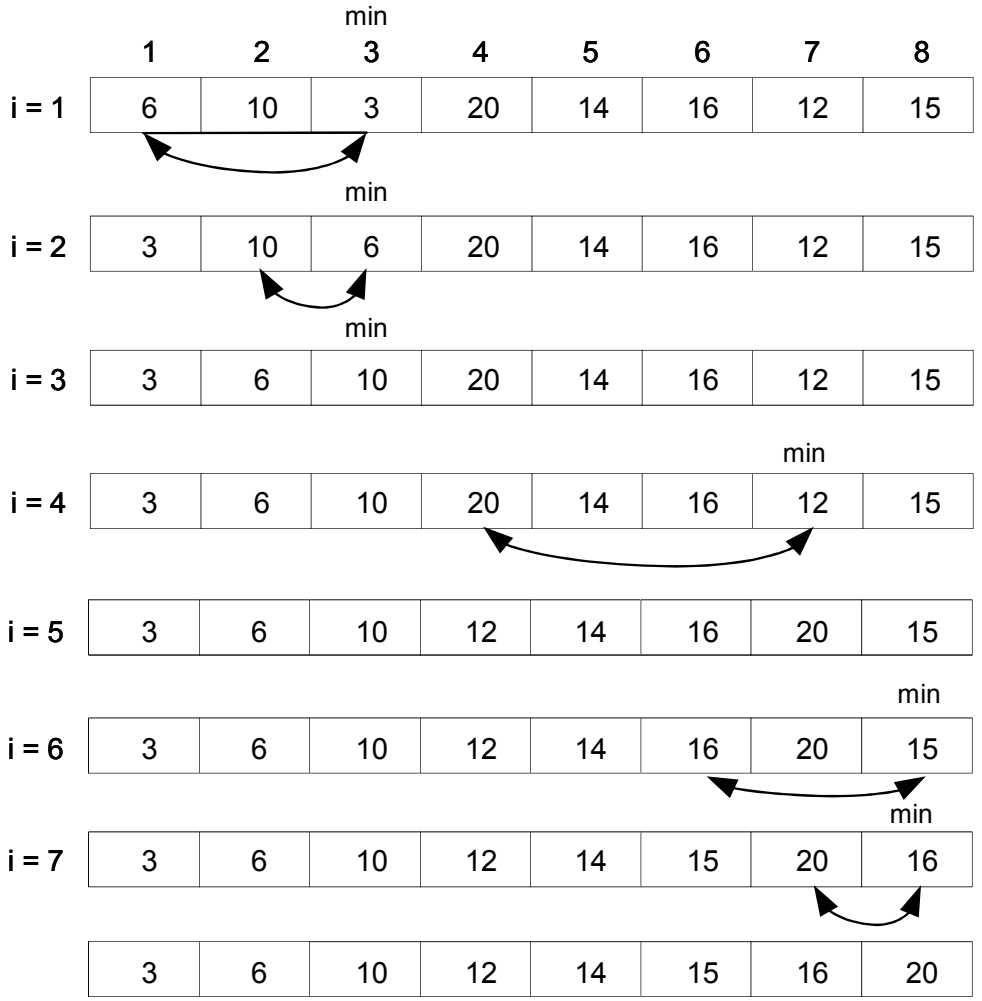

```
procedimiento OrdenaciónIntercambioDirecto(ref vector:v;valor entero:n)
var
   entero : i,j
inicio
   desde i ← 1 hasta n-1 hacer
      desde j ← n hasta i+1 incremento -1 hacer
         si v[j-1] > v[j]entonces
            intercambiar(v[j],v[j-1])
         fin_si
      fin_desde
   fin_desde
fin_procedimiento
```

Ordenación por intercambio directo (II)

```
procedimiento OrdenaciónIntercambioDirecto (ref vector:v;
                                            valor entero : n)
var
   entero : i,j
   lógico : ordenado
inicio
   i ← 0
   repetir
      i ← i + 1
      ordenado ← verdad
      desde j ← n hasta i+1 incremento -1 hacer
         si v[j-1] > v[j]entonces
            intercambiar(v[j],v[j-1])
            ordenado ← falso
         fin_si
      fin_desde
   hasta_que ordenado
fin_procedimiento
```

Ordenación por selección directa

- Busca el primer elemento menor y lo coloca en la primera posición.
- Repite el proceso con los siguientes elementos del array.
- No considera la relación entre los elementos, es decir, no considera que los elementos ya pueden estar colocados.
  - En cada paso sólo se considera uno de ellos.

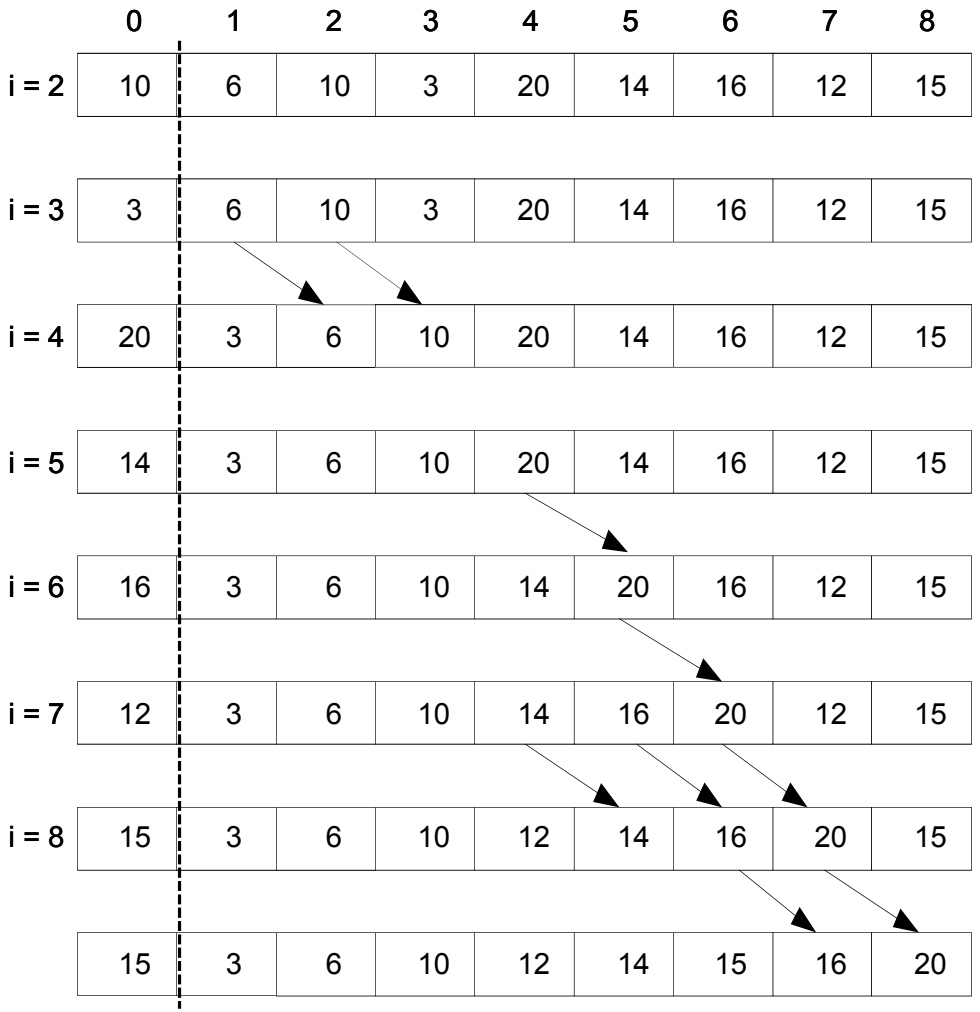


Ordenación por selección directa (II)

```
procedimiento OrdenaciónSelecciónDirecta(ref vector:v;
                                         valor entero : n)
var
   entero : i,j,min
inicio
   desde i ← 1 hasta n-1 hacer
      min ← i
      desde j ← i+1 hasta n hacer
         si v[j] < v[min] entonces
           min ← j
         fin_si
      fin_desde
      intercambiar(v[i],v[min])
   fin_desde
fin_procedimiento
```

Ordenación por inserción directa

- Comprueba la colocación del primer y segundo elemento.
- A partir del elemento 2, se desplazan todos los elementos mayores una posición a la derecha para "abrir hueco".
- Coloca el nuevo elemento en la posición del primer elemento menor o igual.
- Mejora su eficiencia utilizando un centinela.

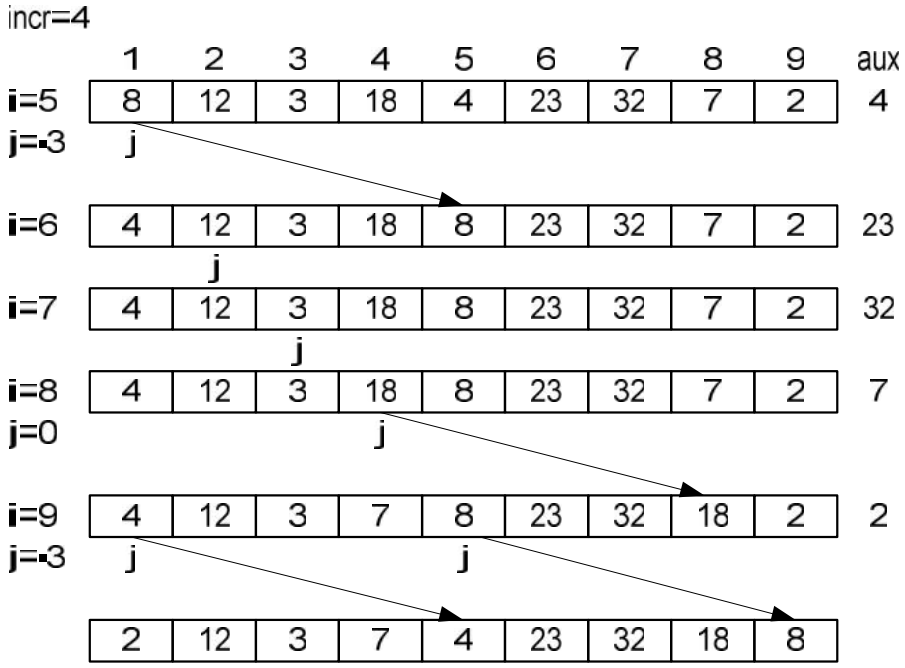


Ordenación por inserción directa (II)

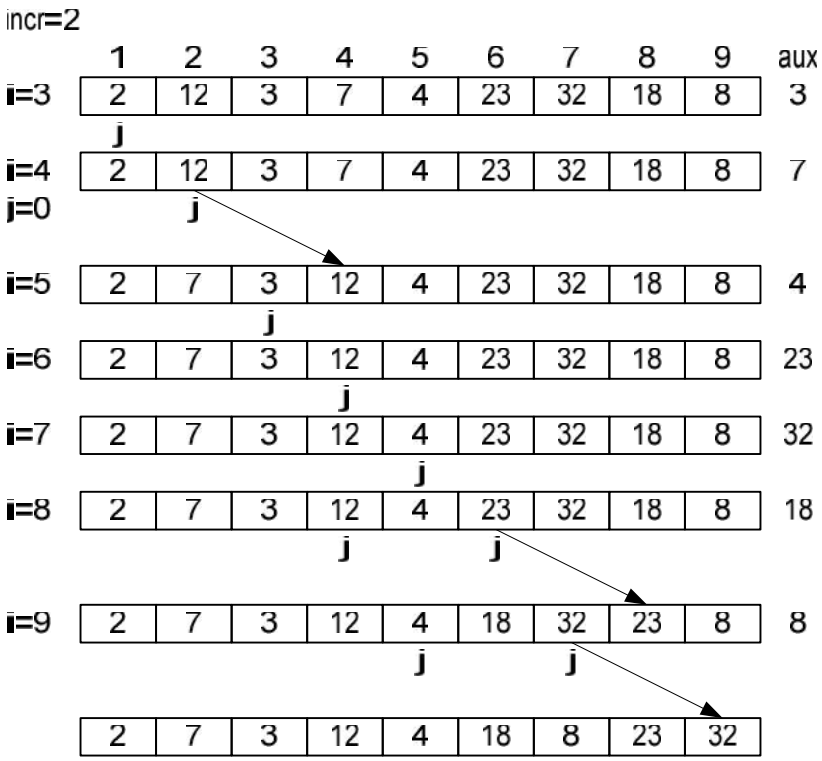
```
procedimiento OrdenaciónInserciónDirecta(ref vector:v;valor entero : n)
//El vector v tiene elementos entre 1 y n
var
   entero : i,j
inicio
   desde i ← 2 hasta n hacer
      //Se almacena el elemento a insertar (v[i]) en la posición 0
      //del array para que actúe como centinela
      v[0] ← v[i]
      j ← i - 1
      //Se desplazan todos los elementos mayores que v[0]
      //y situados a su izquierda una posición a la derecha
      mientras v[j] > v[0] hacer
          v[j+1] ← v[j]
          j ← j - 1
      fin_mientras
      //En la posición siguiente al primer elemento menor o igual
      //se inserta el elemento v[0]
      v[j+1] ← v[0]
   fin_desde
fin_procedimiento
```

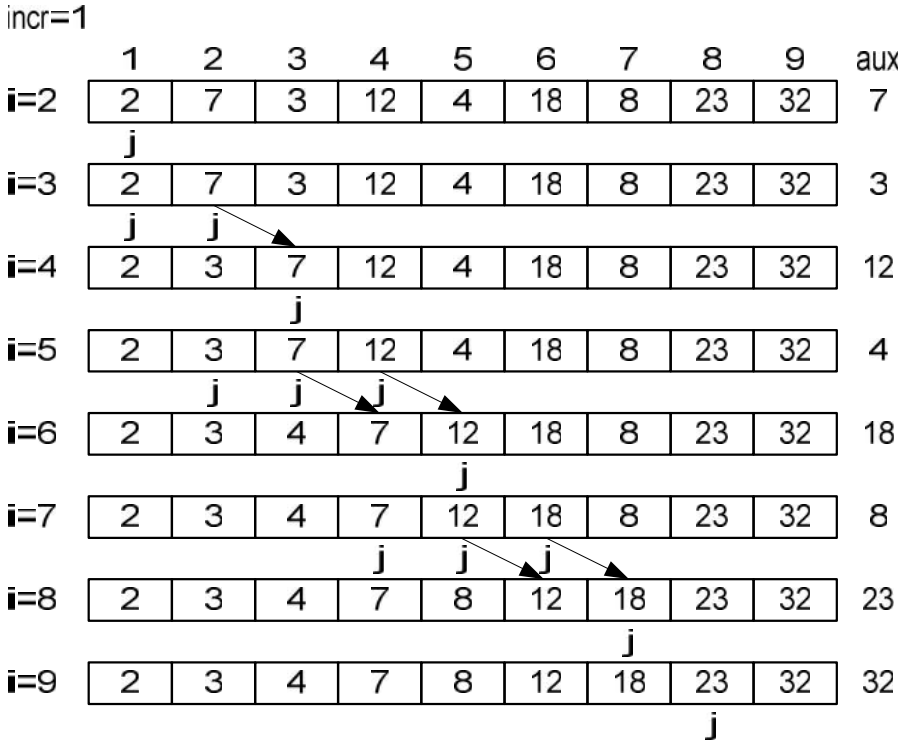
Ordenación por incrementos decrecientes o método de Shell

- Basado en la inserción directa.
- En lugar de comparar elementos adyacentes, la comparación no se realiza siempre con elementos continuos.
  - La distancia entre elementos a comprar decrece a partir de la mitad del número de elementos.
- Repite varias veces el método de inserción directa y en cada repetición modifica la distancia entre los elementos.
- La mejora radica en que cuando la distancia es grande, se tarda poco en realizar la pasada y la lista queda medio ordenada.
- Cuando la distancia es corta, aunque aumenta el número de pasadas, la lista ya está medianamente ordenada.
  - Al final, se compararán elementos adyacentes, pero al estar la lista ordenada se deberán hacer pocos intercambios entre elementos.



Ordenación por incrementos decrecientes (II)





Ordenación por incrementos decrecientes (III)

```
procedimiento OrdenaciónShell(ref vector:v; valor entero : n)
var
   //incr es la separación entre elementos a comparar
   entero : i,j,incr
   tipoElemento : aux
inicio
   //Inicialmente la separación entre elementos a comparar es n/2
   incr ← n div 2
   //Se repite el método de ordenación por inserción mientras
   //que la separación sea mayor que 0
   mientras incr > 0 hacer
     desde i ← incr + 1 hasta n hacer
       aux ← v[i]
       j ← i - incr
      //Se comparan el elemento auxiliar con el situado
      //incr posiciones más a la derecha (elemento v[j])
      mientras j>=1 y v[j]>aux hacer
        v[j+incr] ← v[j]
        j ← j-incr
      fin_mientras
      v[j+incr] ← aux
     fin_desde
     //Una vez que la lista está ordenada entre los elementos
     //situados a incr posiciones, el incremento decrece
     incr ← incr div 2
   fin_mientras
fin_procedimiento
```

Búsqueda

- Localizar un elemento dado entre una lista de ellos para obtener su posición.
  - Normalmente se buscará en arrays paralelos o arrays de registros a partir de un campo clave.
    - ✓ El campo clave normalmente será único.
  - Las funciones de búsqueda para claves únicas devolverán un número entero con la posición del elemento y un centinela (normalmente 0) en el caso de que no lo hayan encontrado.
    - ✓ Para la búsqueda de claves repetidas se deben utilizar métodos alternativos.
      - ○ En ese caso un procedimiento devolverá un array con las posiciones en las que se encuentra ese elemento.

Búsqueda lineal (secuencial)

- Buscar secuencialmente los elementos (uno detrás de otro) hasta encontrarlo o llegar al final de la lista.

```
entero función Buscar(valor vector:v;valor tipoElemento:el;
                      valor entero:n)
var
   entero: i
inicio
   i ← 1
   mientras (el <> v[i]) y (i < n) hacer
      i ← i + 1
   fin_mientras
   si el = v[i] entonces
      devolver(i)
   si_no
      devolver(0)
   fin_si
fin_función
```

Búsqueda lineal con centinela

- Busca desde el último al primer elemento hasta encontrarlo.
  - El elemento a buscar se introduce como centinela en la posición 0 del array.
  - No precisa incluir una condición de encontrado/no encontrado.

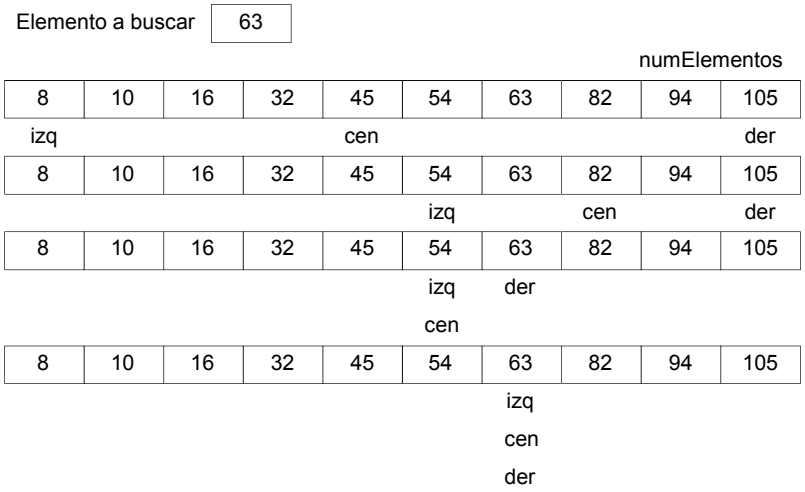
```
entero función Buscar(valor vector:v; valor tipoElemento:el;
                      valor entero:n)
var
   entero: i
inicio
   i ← n
   v[0] ← el
   mientras el <> v[i] hacer //El elemento siempre está
      i ← i - 1
   fin_mientras
   //La función siempre devuelve i
   devolver(i)
fin_función
```

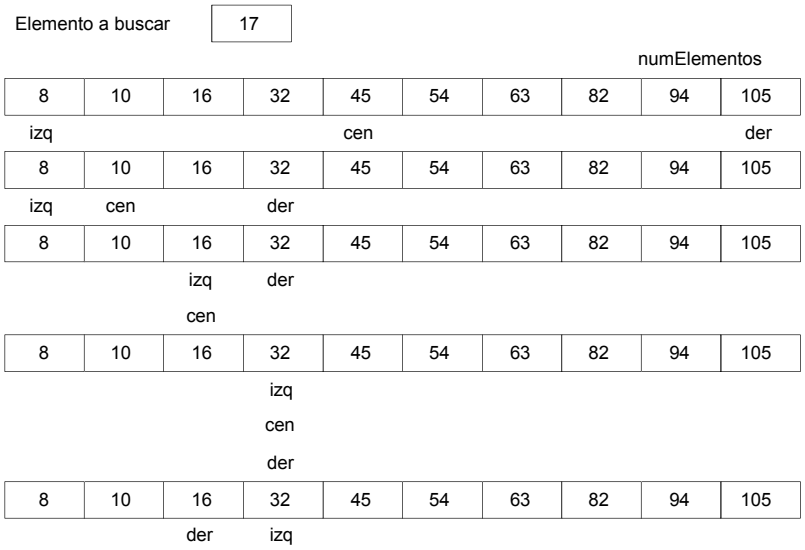
Búsqueda secuencial en un array ordenado

```
entero función Buscar(valor vector:v; valor tipoElemento:el; valor entero:n)
var
   entero: i
inicio
   i ← 1
   mientras (el > v[i]) y (i < n) hacer
      i ← i + 1
   fin_mientras
   si el = v[i] entonces
      devolver(i)
   si_no
      devolver(0)
   fin_si
fin_función
```

Búsqueda binaria o dicotómica

- Precisa que el array esté ordenado.
- Reduce el número máximo de comparaciones a $\log_2(N)$.
- Divide la lista por la mitad.
  - Si el elemento central es menor que el elemento a buscar se descarta los situados a la derecha (`der` ← `cen - 1`).
  - Si el elemento es mayor se descartan los situados a la izquierda (`izq` ← `cen + 1`).
  - El proceso se repite hasta que se encuentra el elemento o es imposible dividir la lista resultante.





Búsqueda binaria o dicotómica (II)

```
entero función Buscar(valor vector:v; valor tipoElemento:el; valor entero:n)
var
   entero: izq, der, cen
inicio
   izq ← 1
   der ← n
   repetir
      cen ← (izq + der) div 2
      si v[cen] > el entonces
         der ← cen - 1
      si_no
         izq ← cen + 1
      fin_si
   hasta_que (v[cen] = el) o (izq > der)
   si v[cen] = el entonces
       devolver(cen)
   si_no
       devolver(0)
   fin_si
fin_función
```

Ordenación por inserción binara

- Como el array queda ordenado entre 1 e i-1, se puede realizar una búsqueda binaria para encontrar la posición del elemento.
  - Una vez localizado el elemento es necesario dejar espacio para el elemento a insertar.

```
procedimiento OrdenaciónInserciónBinaria(ref vector:v ;valor entero : n)
var
   entero : i,j,iz,de,ce
   tipoElemento : aux
inicio
   desde i ← 2 hasta n hacer
      aux ← v[i]
      iz ← 1
      de ← i-1
      mientras iz <= de hacer
         ce ← (iz + de) div 2
         si aux < v[ce] entonces
            de ← ce - 1
         si_no
            iz ← ce + 1
         fin_si
      fin_mientras
      desde j ← i - 1 hasta iz incremento -1 hacer
          v[j+1] ← v[j]
      fin_desde
      v[iz] ← aux
   fin_desde
fin_procedimiento
```

Búsqueda por transformación de claves

- Aplicar una función de transformación de clave (función *hash*) a una clave para que devuelva una posición de almacenamiento válida.

  ```
  dirección ← hash(clave)
  ```

  - La función hash devolvería un número entre 1 y n, siendo n el número de posiciones del vector.
  - Todos los elementos del vector deberían almacenarse utilizando la misma función hash.
- El método de transformación de claves ideal es *`clave → dirección`*.
  - Sólo es posible si las claves son correlativas.
  - Desaprovecha espacio de almacenamiento si las claves no son correlativas.

Métodos de transformación de claves

- Cometidos de la función de transformación de claves:
  - Transformar una clave numérica o alfanumérica en un dirección de almacenamiento válida.
- Deben esparcir los elementos por el espacio lo máximo posible.
  - La dispersión aumenta si aumenta el número de posiciones de almacenamiento (siempre que no aumente el número de claves).
  - Además del espacio necesario para almacenar los datos, se reservará espacio adicional.
    - ✓ Una medida estándar es aproximadamente de un 20%.
- Algunas funciones hash:
  - Aritmética modular
  - Truncamiento.
  - Plegamiento.
  - Mitad del cuadrado.

Métodos de transformación de claves (II)

- Aritmética modular.

  `hash(clave)` → `clave` **`mod`** `n + 1`

  - Devuelve un valor entre 1 y `n`.
  - `n` es un número mayor o igual que el número de elementos a almacenar.
    - ✓ Funciona mejor si n es un número primo.

Clave	hash(clave)
1203	93
6754	89
32	33

Para 100 registros, n podría ser 101

hash(clave) → (clave **mod** 101) + 1

	Clave	Resto de campos
33	32	
89	6754	
93	1203	
120		

Métodos de transformación de claves (III)

- Truncamiento.
  - Extraer n dígitos de la parte central de la clave.
  - n será el número de dígitos máximo de la posición.
  - Se utiliza con claves numéricas grandes.

```
entero función hash(valor entero : clave)
//Para una dirección relativa de 3 dígitos
//Extrae las centenas de millar, decenas de millar y millares
var
   entero : dirección
inicio
   dirección ← clave mod 1000000
   devolver(dirección div 1000 + 1)
fin_función
```

  - Para la clave 53454654, devolvería `clave mod 1000000 div 1000 + 1 = 455`.

Métodos de transformación de claves (IV)

- Plegamiento.
  - Partir la clave en fragmentos de n dígitos y sumarlos, despreciando el acarreo.
  - n será el número de dígitos del número máximo de posiciones.

```
entero función hash(E entero : clave)
//Para una dirección relativa de 3 dígitos
var
   entero : suma,resto
inicio
   suma ← 0
   resto ← clave
   mientras resto > 0 hacer
     suma ← suma + resto mod 1000
     resto ← resto div 1000
   fin_mientras
   devolver(suma mod 1000 + 1)
fin_función
```

Métodos de transformación de claves (V)

- Mitad del cuadrado.
  - Elevar la clave al cuadrado y aplicarle el truncamiento.
- Para claves no numéricas.
  - Convertir la clave alfanumérica en un dígito.
    - ✓ Cambiando cada carácter por su valor según el código ASCII.
    - ✓ Sumando el código ASCII de cada carácter.
  - Aplicar alguno de los métodos anteriores.
- Problema:
  - A no ser que se utilice la función hash `hast(`*`clave`*`)` → *`dirección`* puede que claves distintas generen la misma dirección.
  - Dos claves que generen la misma dirección son **sinónimos**.
  - Cuando se trata de almacenar dos sinónimos se produce una **colisión**.

Tratamiento de colisiones

- Dos problemas:
  - Será necesario averiguar si una posición de almacenamiento ya está ocupada por otra clave.
    - ✓ Se puede inicializar alguno de los campos o la propia clave a un valor centinela (por ejemplo -1).
  - Será necesario buscar un nuevo espacio en el almacenamiento para guardar el sinónimo.
    - ✓ Distintas técnicas.
      - Encadenamiento de colisiones.
      - Llevar las colisiones a una zona espacial (direccionamiento a zona de colisiones).
      - Almacenar los sinónimos lo más cerca posible de su posición original (direccionamiento a vacío).

Tratamiento de colisiones (II)

- Al intentar guardar el elemento con clave 1363 se comprobaría que está ocupado y se iría a la primera dirección libre (posición 45).
- Al intentar guardar la clave 1079 (dirección 120), se intentaría ir a la primera posición libre (121).
  - Como está fuera del espacio de almacenamiento se guardaría en la posición 1.

Clave	hash(clave)
1203	4
6754	35
32	33
8683	44
839	120
1363	44
1079	120

hash(clave) → clave **mod** 120 + 1

	Clave	Resto de campos
1	1079	
	-1	
4	1203	
	-1	
33	32	
	-1	
35	6754	
	-1	
	-1	
44	8683	
45	1363	
	-1	
	-1	
120	839	

Búsqueda por transformación de claves

```
//Se desea buscar el registro con clave claveBuscada
dirección ← hash(claveBuscada)
si v[dirección].clave < 0 entonces
   //El elemento no se encuentra
si_no
   si v[dirección].clave <> claveBuscada entonces
      //Puede que se trate de un sinónimo
      //Se busca entre las posiciones siguientes
      //hasta que se encuentra o hasta hallar un hueco vacío
      repetir
         dirección ← dirección mod numElementos + 1
      hasta_que (v[dirección].clave = claveBuscada) o v[dirección].clave < 0
   fin_si
fin_si
si v[dirección].clave = claveBuscada entonces
   //El elemento se encuentra en la posición dirección
si_no
   //El elemento no se encuentra
fin_si

entero función hash(valor entero: clave)
inicio
   devolver(clave mod numElementos +1)
fin_funcion
```

Intercalación

- Tomar dos o más arrays de entrada ordenados y obtener un tercer array de salida también ordenado.
  - Se compara el primer elemento de los dos arrays de entrada.
  - Se selecciona el menor de ellos que será el primer elemento del array de salida.
  - Se avanza el índice del array del que se ha seleccionado el elemento.
  - El proceso se repite hasta que se termina alguno de los arrays.
  - Se vuelca el array que no se haya terminado al array de salida.

Intercalación (II)

```
procedimiento Intercalación(valor vector:A,B;ref vector:C; valor entero: m,n)
var
   //i será el índice de A, j el de B y k el de C
   entero : i,j,k
inicio
   i ← 1
   j ← 1
   k ← 1
   //Mientras no se acabe alguno de los arrays de entrada
   mientras (i <= M) y (j <= N) hacer
      si A[i] < B[j] entonces
         //Se selecciona un elemento de A y se inserta en C
         C[k] ← A[i]
         //Se desplaza el índice del array A
         i ← i + 1
      si_no
         //Se selecciona un elemento de B y se inserta en C
         C[k] ← B[j]
         //Se desplaza el índice del array B
         j ← j + 1
      fin_si
      //Se desplaza el índice del array de salida
      k ← k + 1
   fin_mientras
```

Intercalación (III)

```
   //Si se ha llegado al final del array B se vuelca todo
   //el contenido que queda de A en el array de salida
   mientras i <= M hacer
      C[k] ← A[i]
      i ← i + 1
      k ← k + 1
   fin_mientras
   //Si se ha llegado al final del array A se vuelca todo
   //el contenido que queda de B en el array de salida
   mientras j <= N hacer
      C[k] ← B[j]
      j ← j + 1
      k ← k + 1
   fin_mientras
fin_procedimiento
```

Ejercicios

1. En una tabla se almacenan las ventas realizadas por n vendedores en cada uno de los 12 meses del año.
   - Generar un array de registros en el que se almacene en número de vendedor (número de la fila dónde se encuentra) y el total de ventas realizadas por el vendedor en el año.
   - Ordenar el array de registros de menor a mayor por el total de ventas realizadas.
2. Se desea obtener la tabla de puntuación en una liga de futbolín en la que compiten 10 equipos a una sola vuelta. Los resultados se almacenarán en una tabla de dos dimensiones, de forma que cada elemento i,j de la tabla guarda los tantos que ha marcado el equipo i al equipo j en el encuentro. Por ejemplo, un 8 en la posición 2,6 significa que en el encuentro entre el equipo 2 y el equipo 6, el equipo 2 ha metido 8 tantos al equipo 6.

   Los nombres de los equipos y la puntuación obtenida por cada uno de ellos se almacenarán en un array de registros de 10 posiciones (una por equipo) que almacenará el nombre del equipo y la puntuación.
   - Declarar las estructuras de datos necesarias para realizar la aplicación.
   - Diseñar un módulo que permita introducir por teclado el nombre de los equipos y almacenarlo en el array de registros.
   - Diseñar un módulo que permita rellenar la tabla con los tantos obtenidos por los equipos.
   - Diseñar un módulo que permita obtener la puntuación de cada equipo (3 puntos por partido ganado, 1 por partido empatado y 0 por partido perdido).
   - Diseñar un módulo que permita obtener la clasificación de la liga.

Ejercicios (II)

3. Se tiene un array de registros con los participantes en un meeting de atletismo (dorsal y país). En otro array de registros están los países y tres campos para las medallas de oro, plata y bronce. Una tabla contiene los resultados de las pruebas (una fila por prueba, cada columna indica la posición del atleta). A partir de esto se desea obtener el medallero ordenado por medallas de oro, plata y bronce.